下载APP

02｜Context：请求控制器，让每个请求都在掌控之中

2021-09-15 叶剑峰

《手把手带你写一个Web框架》 课程介绍 >

讲述：叶剑峰

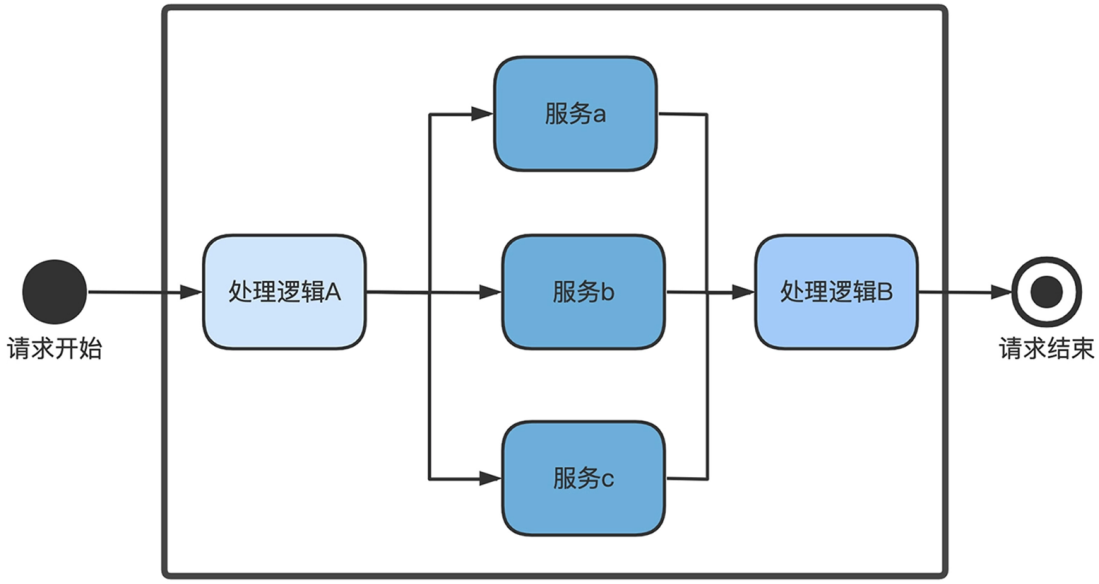
时长 19:13 大小 17.61M

你好，我是轩脉刃。

上一讲我们使用 net/http 搭建了一个最简单的 HTTP 服务，为了帮你理解服务启动逻辑，我用思维导图梳理了主流程，如果你还有不熟悉，可以再回顾一下。

今天我将带你进一步丰富我们的框架，添加上下文 Context 为请求设置超时时间。

从主流程中我们知道（第三层关键结论），HTTP 服务会为每个请求创建一个 Goroutine 进行服务处理。在服务处理的过程中，有可能就在本地执行业务逻辑，也有可能再去下游服务获取数据。如下图，本地处理逻辑 A/B、下游服务 a/b/c，会形成一个标准的树形逻辑链条。



在这个逻辑链条中，每个本地处理逻辑，或者下游服务请求节点，都有可能存在超时问题。**而对于 HTTP 服务而言，超时往往是造成服务不可用、甚至系统瘫痪的罪魁祸首。**

系统瘫痪也就是我们俗称的雪崩，某个服务的不可用引发了其他服务的不可用。比如上图中，如果服务 a 超时，导致请求处理缓慢甚至不可用，加剧了 Goroutine 堆积，同时也造成了服务 b 和服务 c 的瞬时请求加大，导致 b 和 c 的服务都不可用，整个系统瘫痪。怎么办？

最有效的方法就是从源头上控制一个请求的“最大处理时长”，所以，对于一个 Web 框架而言，“超时控制”能力是必备的。今天我们就用 Context 为框架增加这个能力。

context 标准库设计思路

如何控制超时，官方是有提供 context 标准库作为解决方案的，但是由于标准库的功能并不够完善，一会我们会基于标准库，来根据需求自定义框架的 Context。所以理解其背后的设计思路就可以了。

为了防止雪崩，context 标准库的解决思路是：**在整个树形逻辑链条中，用上下文控制器 Context，实现每个节点的信息传递和共享。**

具体操作是：用 Context 定时器为整个链条设置超时时间，时间一到，结束事件被触发，链条中正在处理的服务逻辑会监听到，从而结束整个逻辑链条，让后续操作不再进行。

明白操作思路之后，我们深入 context 标准库看看要对应具备哪些功能。

按照上一讲介绍的了解标准库的方法，我们先通过 `go doc context | grep "^func"` 看提供了哪些库函数（function）：

复制代码

```
// 创建退出 Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc){}
// 创建有超时时间的 Context
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc){
// 创建有截止时间的 Context
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc){}
```

其中，WithCancel 直接创建可以操作退出的子节点，WithTimeout 为子节点设置了超时时间（还有多少时间结束），WithDeadline 为子节点设置了结束时间线（在什么时间结束）。

但是这只是表层功能的不同，其实这三个库函数的**本质是一致的**。怎么理解呢？

我们先通过 `go doc context | grep "^type"` ，搞清楚 Context 的结构定义和函数句柄，再来解答这个问题。

复制代码

```
type Context interface {
    // 当 Context 被取消或者到了 deadline，返回一个被关闭的 channel
    Done() <-chan struct{}
    ...
}

//函数句柄
type CancelFunc func()
```

这个库虽然不大，但是设计感强，比较抽象，并不是很好理解。所以这里，我把 Context 的其他字段省略了。现在，我们只理解核心的 Done() 方法和 CancelFunc 这两个函数就

可以了。

在树形逻辑链条上，**一个节点其实有两个角色：一是下游树的管理者；二是上游树的被管理者**，那么就对应需要有两个能力：

一个是能让整个下游树结束的能力，也就是函数句柄 CancelFunc；

另外一个是在上游树结束的时候被通知的能力，也就是 Done() 方法。同时因为通知是需要不断监听的，所以 Done() 方法需要通过 channel 作为返回值让使用方进行监听。

看官方代码示例：

复制代码

```
package main

import (
  "context"
  "fmt"
  "time"
)

const shortDuration = 1 * time.Millisecond

func main() {
    // 创建截止时间
  d := time.Now().Add(shortDuration)
    // 创建有截止时间的 Context
  ctx, cancel := context.WithDeadline(context.Background(), d)
  defer cancel()

    // 使用 select 监听 1s 和有截止时间的 Context 哪个先结束
  select {
  case <-time.After(1 * time.Second):
    fmt.Println("overslept")
  case <-ctx.Done():
    fmt.Println(ctx.Err())
  }

}
```

主线程创建了一个 1 毫秒结束的定时器 Context，在定时器结束的时候，主线程会通过 Done() 函数收到事件结束通知，然后主动调用函数句柄 cancelFunc 来通知所有子 Context 结束（这个例子比较简单没有子 Context）。

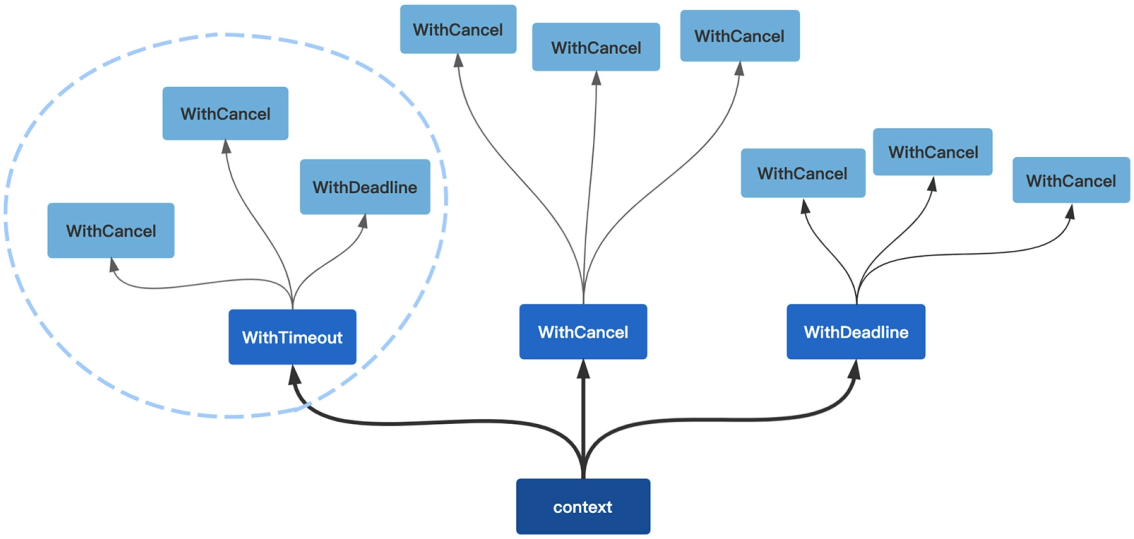
我打个更形象的比喻，CancelFunc 和 Done 方法就像是电话的话筒和听筒，话筒 CancelFunc，用来告诉管辖范围内的所有 Context 要进行自我终结，而通过监听听筒 Done 方法，我们就能听到上游父级管理者的终结命令。

总之，**CancelFunc 是主动让下游结束，而 Done 是被上游通知结束**。

搞懂了具体实现方法，我们回过头来看这三个库函数 WithCancel / WithDeadline / WithTimeout 就很好理解了。

它们的本质就是“通过定时器来自动触发终结通知”，WithTimeout 设置若干秒后通知触发终结，WithDeadline 设置未来某个时间点触发终结。

对应到 Context 代码中，它们的功能就是：**为一个父节点生成一个带有 Done 方法的子节点，并且返回子节点的 CancelFunc 函数句柄**。



极客时间

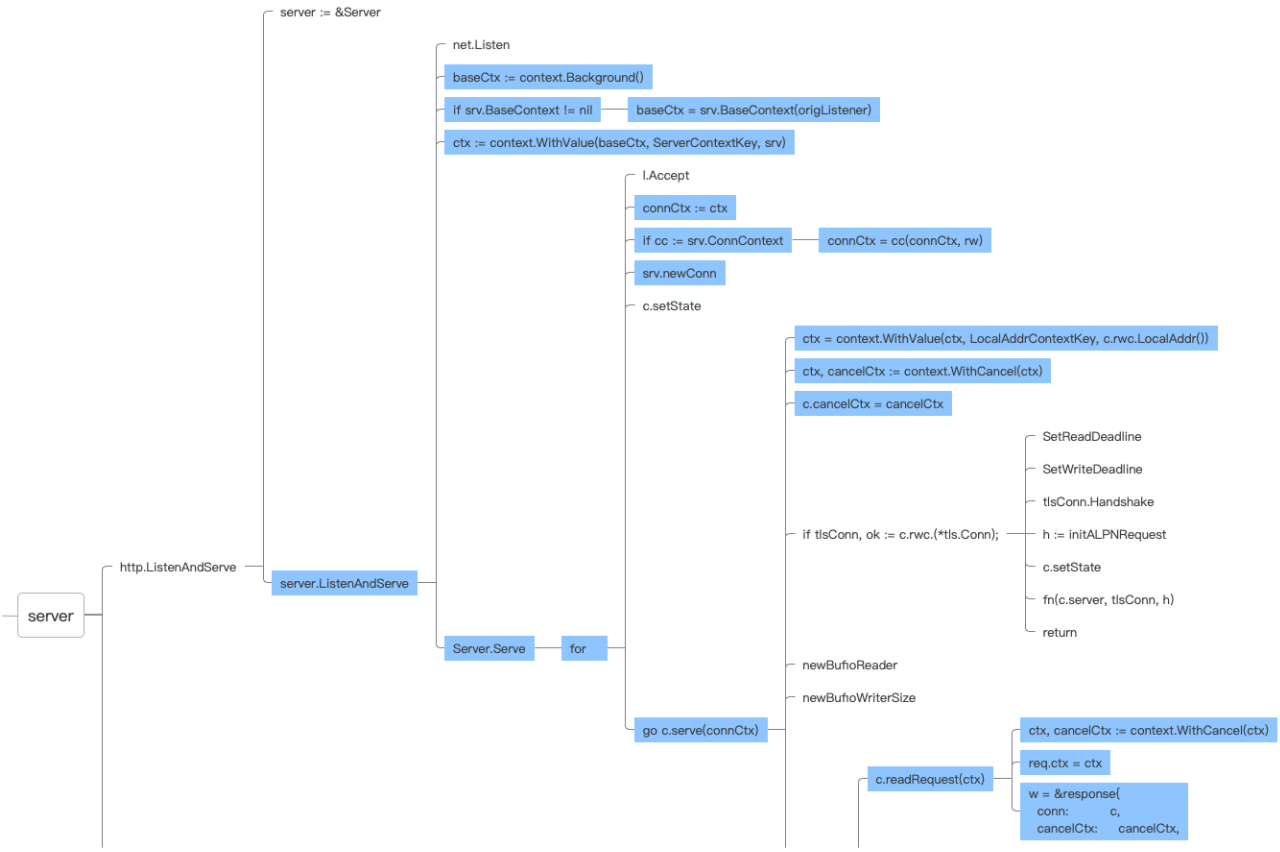
我们用一张图来辅助解释一下，Context 的使用会形成一个树形结构，下游指的是树形结构中的子节点及所有子节点的子树，而上游指的是当前节点的父节点。比如图中圈起来的部分，当 WithTimeout 调用 CancelFunc 的时候，所有下游的 With 系列产生的 Context 都会从 Done 中收到消息。

Context 是怎么产生的

现在我们已经了解标准库 context 的设计思路了，在开始写代码之前，我们还要把 Context 放到 net/http 的主流程逻辑中，其中有两个问题要搞清楚：**Context 在哪里产生？它的上下游逻辑是什么？**

要回答这两个问题，可以用我们在上一讲介绍的思维导图方法，因为主流程已经拎清楚了，现在你只需要把其中 Context 有关的代码再详细过一遍，然后在思维导图上标记出来就可以了。

这里，我已经把 Context 的关键代码都用蓝色背景做了标记，你可以检查一下自己有没有标漏。

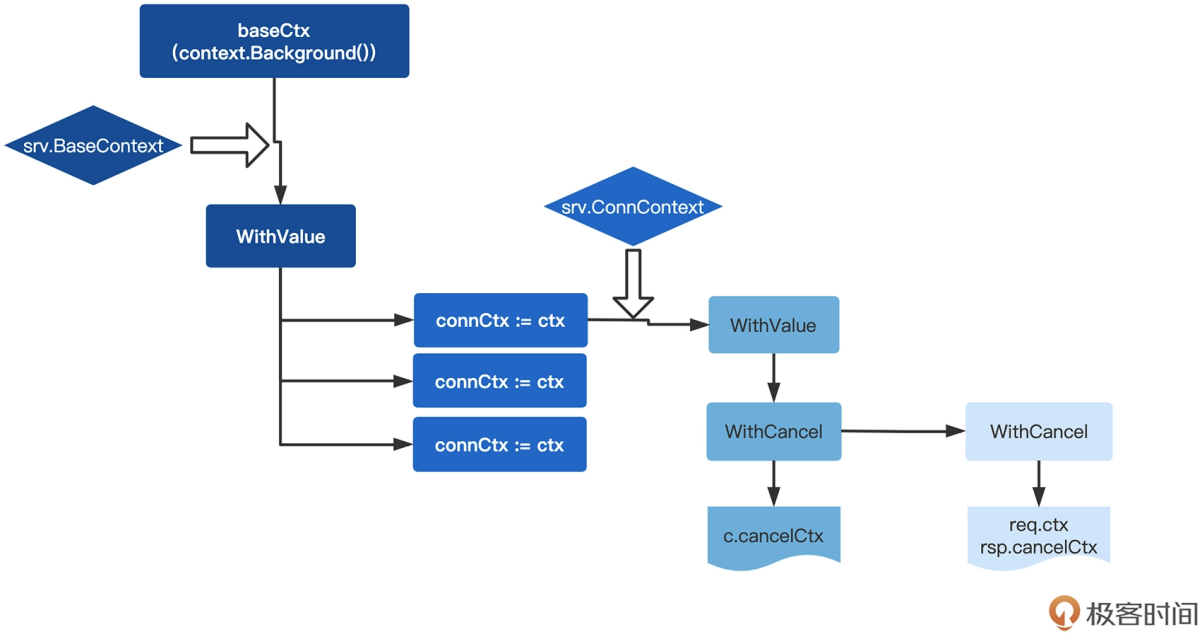


照旧看图梳理代码流程，来看蓝色部分，从前到后的层级梳理就不再重复讲了，我们看关键位置。

从图中最后一层的代码 req.ctx = ctx 中看到，每个连接的 Context 最终是放在 request 结构体中的。

而且这个时候， Context 已经有多层父节点。因为，在代码中，每执行一次 WithCancel、WithValue，就封装了一层 Context，我们通过这一张流程图能清晰看到最

终 Context 的生成层次。



你发现了吗，**其实每个连接的 Context 都是基于 baseContext 复制来的**。对应到代码中就是，在为某个连接开启 Goroutine 的时候，为当前连接创建了一个 connContext，这个 connContext 是基于 server 中的 Context 而来，而 server 中 Context 的基础就是 baseContext。

所以，Context 从哪里产生这个问题，我们就解决了，但是如果我们想要对 Context 进行必要的修改，还要从上下游逻辑中，找到它的修改点在哪里。

生成最终的 Context 的流程中，net/http 设计了**两处可以注入修改**的地方，都在 Server 结构里面，一处是 BaseContext，另一处是 ConnContext。

> BaseContext 是整个 Context 生成的源头，如果我们不希望使用默认的 context.Backgroud()，可以替换这个源头。
>
> 而在每个连接生成自己要使用的 Context 时，会调用 ConnContext ，它的第二个参数是 net.Conn，能让我们对某些特定连接进行设置，比如要针对性设置某个调用 IP。

这两个函数的定义我写在下面的代码里了，展示一下，你可以看看。

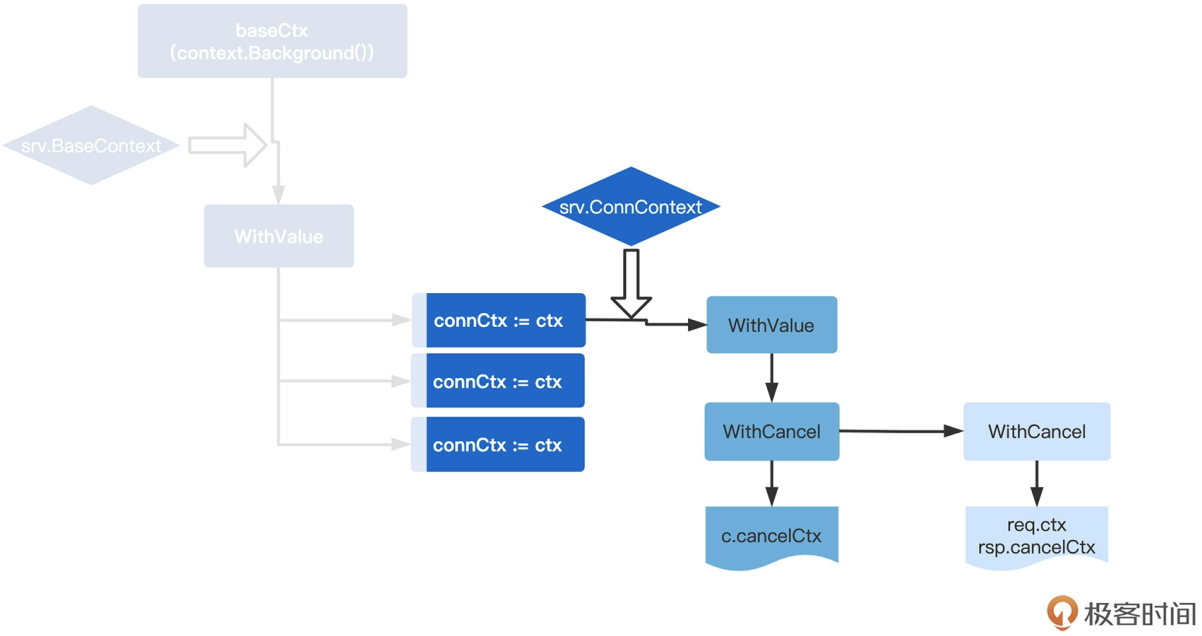
复制代码

```
type Server struct {
  ...

    // BaseContext 用来为整个链条创建初始化 Context
    // 如果没有设置的话，默认使用 context.Background()
  BaseContext func(net.Listener) context.Context{}

    // ConnContext 用来为每个连接封装 Context
    // 参数中的 context.Context 是从 BaseContext 继承来的
  ConnContext func(ctx context.Context, c net.Conn) context.Context{}
    ...
}
```

最后，我们回看一下 req.ctx 是否能感知连接异常。



是可以的，因为链条中一个父节点为 CancelContext，其 cancelFunc 存储在代表连接的 conn 结构中，连接异常的时候，会触发这个函数句柄。

好，讲完 context 库的核心设计思想，以及在 net/http 的主流程逻辑中嵌入 context 库的关键实现，我们现在心中有图了，就可以撸起袖子开始写框架代码了。

你是不是有点疑惑，为啥要自己先理解一遍 context 标准库的生成流程，咱们直接动手干不是更快？有句老话说得好，磨刀不误砍柴功。

我们确实是要自定义，不是想直接使用标准库的 Context，因为它完全是标准库 Context 接口的实现，只能控制链条结束，封装性并不够。但是只有先搞清楚了 context 标准库的设计思路，才能精准确定自己能怎么改、改到什么程度合适，下手的时候才不容易懵。

下面我们就基于刚才讲的设计思路，从封装自己的 Context 开始，写今天的核心逻辑，也就是为单个请求设置超时，最后考虑一些边界场景，并且进行优化。

我们还是再拉一个分支 geekbang/02，接着上一节课的代码结构，在框架文件夹中封装一个自己的 Context。

封装一个自己的 Context

在框架里，我们需要有更强大的 Context，除了可以控制超时之外，常用的功能比如获取请求、返回结果、实现标准库的 Context 接口，也都要有。

我们首先来设计提供获取请求、返回结果功能。

先看一段未封装自定义 Context 的控制器代码：

复制代码

```
// 控制器
func Foo1(request *http.Request, response http.ResponseWriter) {
  obj := map[string]interface{}{
    "data":   nil,
  }
    // 设置控制器 response 的 header 部分
  response.Header().Set("Content-Type", "application/json")

    // 从请求体中获取参数
  foo := request.PostFormValue("foo")
  if foo == "" {
    foo = "10"
  }
  fooInt, err := strconv.Atoi(foo)
  if err != nil {
    response.WriteHeader(500)
    return
  }
    // 构建返回结构
  obj["data"] = fooInt
  byt, err := json.Marshal(obj)
  if err != nil {
```

```
23      response.WriteHeader(500)
24      return
25    }
26      // 构建返回状态，输出返回结构
27    response.WriteHeader(200)
28    response.Write(byt)
29    return
30  }
```

这段代码重点是操作调用了 http.Request 和 http.ResponseWriter ，实现 WebService 接收和处理协议文本的功能。但这两个结构提供的接口粒度太细了，需要使用者非常熟悉这两个结构的内部字段，比如 response 里设置 Header 和设置 Body 的函数，用起来肯定体验不好。

如果我们能**将这些内部实现封装起来，对外暴露语义化高的接口函数**，那么我们这个框架的易用性肯定会明显提升。什么是好的封装呢？再看这段有封装的代码：

复制代码

```
// 控制器
func Foo2(ctx *framework.Context) error {
  obj := map[string]interface{}{
    "data":   nil,
  }
    // 从请求体中获取参数
   fooInt := ctx.FormInt("foo", 10)
    // 构建返回结构
  obj["data"] = fooInt
    // 输出返回结构
  return ctx.Json(http.StatusOK, obj)
}
```

你可以明显感受到封装性高的 Foo2 函数，更优雅更易读了。首先它的代码量更少，而且语义性也更好，近似对业务的描述：从请求体中获取 foo 参数，并且封装为 Map，最后 JSON 输出。

思路清晰了，所以这里可以将 request 和 response 封装到我们自定义的 Context 中，对外提供请求和结果的方法，我们把这个 Context 结构写在框架文件夹的 context.go 文件中：

复制代码

```
// 自定义 Context
type Context struct {
  request        *http.Request
  responseWriter http.ResponseWriter
  ...
}
```

对 request 和 response 封装的具体实现，我们到第五节课封装的时候再仔细说。

然后是第二个功能，标准库的 Context 接口。

标准库的 Context 通用性非常高，基本现在所有第三方库函数，都会根据官方的建议，将第一个参数设置为标准 Context 接口。所以我们封装的结构只有实现了标准库的 Context，才能方便直接地调用。

到底有多方便，我们看使用示例：

复制代码

```
func Foo3(ctx *framework.Context) error {
  rdb := redis.NewClient(&redis.Options{
    Addr:     "localhost:6379",
    Password: "", // no password set
    DB:       0,  // use default DB
  })

  return rdb.Set(ctx, "key", "value", 0).Err()
}

```

这里使用了 go-redis 库，它每个方法的参数中都有一个标准 Context 接口，这让我们能将自定义的 Context 直接传递给 rdb.Set。

所以在我们的框架上实现这一步，只需要调用刚才封装的 request 中的 Context 的标准接口就行了，很简单，我们继续在 context.go 中进行补充：

复制代码

```
func (ctx *Context) BaseContext() context.Context {
  return ctx.request.Context()
}

```

```
5 func (ctx *Context) Done() <-chan struct{} {
6   return ctx.BaseContext().Done()
7 }
```

这里举例了两个 method 的实现，其他的都大同小异就不在文稿里展示，你可以先自己写，然后对照我放在GitHub上的完整代码检查一下。

自己封装的 Context 最终需要提供四类功能函数：

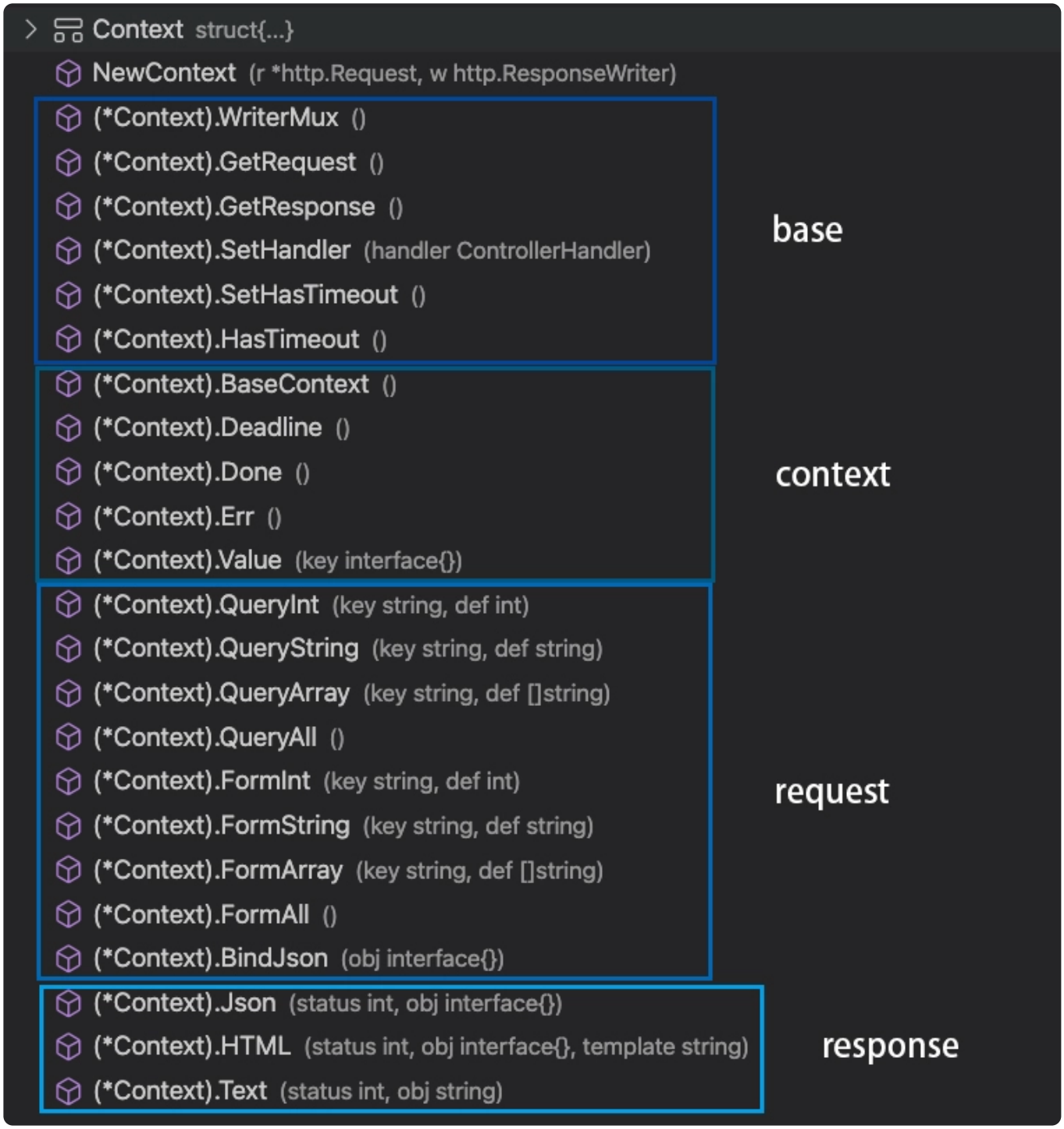
base 封装基本的函数功能，比如获取 http.Request 结构

context 实现标准 Context 接口

request 封装了 http.Request 的对外接口

response 封装了 http.ResponseWriter 对外接口

完成之后，使用我们的 IDE 里面的结构查看器（每个 IDE 显示都不同），就能查看到如下的函数列表：



有了我们自己封装的 Context 之后，控制器就非常简化了。把框架定义的 ControllerHandler 放在框架目录下的 controller.go 文件中：

复制代码

```
type ControllerHandler func(c *Context) error
```

把处理业务的控制器放在业务目录下的 controller.go 文件中：

复制代码

```
func FooControllerHandler(ctx *framework.Context) error {
  return ctx.Json(200, map[string]interface{}{
    "code": 0,
  })
}
```

参数只有一个 framework.Context，是不是清爽很多，这都归功于刚完成的自定义 Context。

为单个请求设置超时

上面我们封装了自定义的 Context，从设计层面实现了标准库的 Context。下面回到我们这节课核心要解决的问题，为单个请求设置超时。

如何使用自定义 Context 设置超时呢？结合前面分析的标准库思路，我们三步走完成：

1. 继承 request 的 Context，创建出一个设置超时时间的 Context；
2. 创建一个新的 Goroutine 来处理具体的业务逻辑；
3. 设计事件处理顺序，当前 Goroutine 监听超时时间 Contex 的 Done() 事件，和具体的业务处理结束事件，哪个先到就先处理哪个。

理清步骤，我们就可以在业务的 controller.go 文件中完成业务逻辑了。**第一步生成一个超时的 Context**：

复制代码

```
durationCtx, cancel := context.WithTimeout(c.BaseContext(), time.Duration(1*ti
// 这里记得当所有事情处理结束后调用 cancel，告知 durationCtx 的后续 Context 结束
defer cancel()
```

这里为了最终在浏览器做验证，我设置超时事件为 1s，这样最终验证的时候，最长等待 1s 就可以知道超时是否生效。

第二步创建一个新的 Goroutine 来处理业务逻辑：

复制代码

```
finish := make(chan struct{}, 1)
```

```
2
3  go func() {
4      ...
5      // 这里做具体的业务
6      time.Sleep(10 * time.Second)
7          c.Json(200, "ok")
8          ...
9          // 新的 goroutine 结束的时候通过一个 finish 通道告知父 goroutine
10     finish <- struct{}{}
11 }()
```

为了最终的验证效果，我们使用 time.Sleep 将新 Goroutine 的业务逻辑事件人为往后延迟了 10s，再输出“ok”，这样最终验证的时候，效果比较明显，因为前面的超时设置会在 1s 生效了，浏览器就有表现了。

到这里我们这里先不急着进入第三步，还有错误处理情况没有考虑到位。这个新创建的 Goroutine 如果出现未知异常怎么办？需要我们额外捕获吗？

其实在 Golang 的设计中，每个 Goroutine 都是独立存在的，父 Goroutine 一旦使用 Go 关键字开启了一个子 Goroutine，父子 Goroutine 就是平等存在的，他们互相不能干扰。而在异常面前，所有 Goroutine 的异常都需要自己管理，不会存在父 Goroutine 捕获子 Goroutine 异常的操作。

所以切记：在 Golang 中，每个 Goroutine 创建的时候，我们要使用 defer 和 recover 关键字为当前 Goroutine 捕获 panic 异常，并进行处理，否则，任意一处 panic 就会导致整个进程崩溃！

这里你可以标个重点，面试会经常被问到。

搞清楚这一点，**我们回看第二步，做完具体业务逻辑就结束是不行的，还需要处理 panic。**所以这个 Goroutine 应该要有两个 channel 对外传递事件：

复制代码

```
1 // 这个 channal 负责通知结束
2 finish := make(chan struct{}, 1)
3 // 这个 channel 负责通知 panic 异常
4 panicChan := make(chan interface{}, 1)
5
6 go func() {
```

```
        // 这里增加异常处理
    defer func() {
      if p := recover(); p != nil {
        panicChan <- p
      }
    }()
    // 这里做具体的业务
    time.Sleep(10 * time.Second)
        c.Json(200, "ok")
        ...
        // 新的 goroutine 结束的时候通过一个 finish 通道告知父 goroutine
    finish <- struct{}{}
}()
```

现在第二步才算完成了，我们继续写第三步监听。使用 select 关键字来监听三个事件：异常事件、结束事件、超时事件。

复制代码

```
  select {
    // 监听 panic
  case p := <-panicChan:
    ...
        c.Json(500, "panic")
    // 监听结束事件
  case <-finish:
    ...
        fmt.Println("finish")
    // 监听超时事件
  case <-durationCtx.Done():
    ...
        c.Json(500, "time out")
  }
```

接收到结束事件，只需要打印日志，但是，在接收到异常事件和超时事件的时候，我们希望告知浏览器前端“异常或者超时了”，所以会使用 c.Json 来返回一个字符串信息。

三步走到这里就完成了对某个请求的超时设置，你可以通过 go build、go run 尝试启动下这个服务。如果你在浏览器开启一个请求之后，浏览器不会等候事件处理 10s，而在等待我们设置的超时事件 1s 后，页面显示“time out”就结束这个请求了，就说明我们为某个事件设置的超时生效了。

边界场景

到这里，我们的超时逻辑设置就结束且生效了。但是，这样的代码逻辑只能算是及格，为什么这么说呢？因为它并没有覆盖所有的场景。

我们的代码逻辑要再严谨一些，**把边界场景也考虑进来**。这里有两种可能：

1. 异常事件、超时事件触发时，需要往 responseWriter 中写入信息，这个时候如果有其他 Goroutine 也要操作 responseWriter，会不会导致 responseWriter 中的信息出现乱序？
2. 超时事件触发结束之后，已经往 responseWriter 中写入信息了，这个时候如果有其他 Goroutine 也要操作 responseWriter， 会不会导致 responseWriter 中的信息重复写入？

你先分析第一个问题，是很有可能出现的。方案不难想到，我们要保证在事件处理结束之前，不允许任何其他 Goroutine 操作 responseWriter，这里可以使用一个锁（sync.Mutex）对 responseWriter 进行写保护。

在框架文件夹的 context.go 中对 Context 结构进行一些设置：

复制代码

```
type Context struct {
  // 写保护机制
  writerMux  *sync.Mutex
}

// 对外暴露锁
func (ctx *Context) WriterMux() *sync.Mutex {
  return ctx.writerMux
}
```

在刚才写的业务文件夹 controller.go 中也进行对应的修改：

复制代码

```
func FooControllerHandler(c *framework.Context) error {
  ...
    // 请求监听的时候增加锁机制
  select {
  case p := <-panicChan:
    c.WriterMux().Lock()
```

```
    defer c.WriterMux().Unlock()
    ...
    c.Json(500, "panic")
  case <-finish:
        ...
    fmt.Println("finish")
  case <-durationCtx.Done():
    c.WriterMux().Lock()
    defer c.WriterMux().Unlock()
    c.Json(500, "time out")
    c.SetTimeout()
  }
  return nil
}
```

那第二个问题怎么处理，我提供一个方案。我们可以**设计一个标记**，当发生超时的时候，设置标记位为 true，在 Context 提供的 response 输出函数中，先读取标记位；当标记位为 true，表示已经有输出了，不需要再进行任何的 response 设置了。

同样在框架文件夹中修改 context.go：

复制代码

```
type Context struct {
    ...
  // 是否超时标记位
  hasTimeout bool
  ...
}

func (ctx *Context) SetHasTimeout() {
  ctx.hasTimeout = true
}

func (ctx *Context) Json(status int, obj interface{}) error {
  if ctx.HasTimeout() {
    return nil
  }
  ...
}
```

在业务文件夹中修改 controller.go：

复制代码

```
func FooControllerHandler(c *framework.Context) error {
```

```
  ...
  select {
  case p := <-panicChan:
    ...
  case <-finish:
    fmt.Println("finish")
  case <-durationCtx.Done():
    c.WriterMux().Lock()
    defer c.WriterMux().Unlock()
    c.Json(500, "time out")
        // 这里记得设置标记为
    c.SetHasTimeout()
  }
  return nil
}
```

好了，到了这里，我们就完成了请求超时设置，并且考虑了边界场景。

剩下的验证部分，我们写一个简单的路由函数，将这个控制器路由在业务文件夹中创建一个 route.go:

复制代码

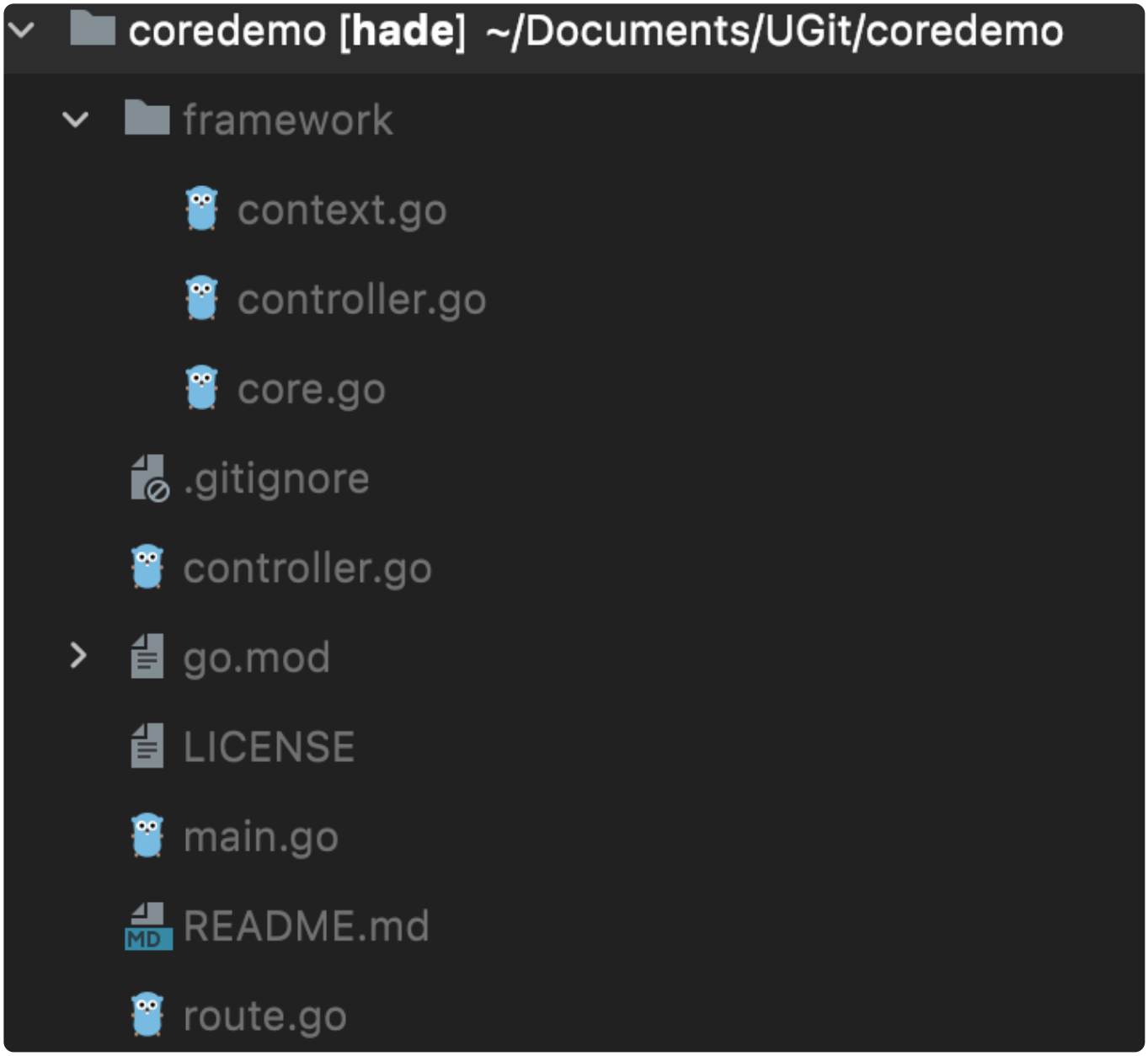
```
func registerRouter(core *framework.Core) {
  // 设置控制器
   core.Get("foo", FooControllerHandler)
}
```

并修改 main.go：

复制代码

```
func main() {
   ...
   // 设置路由
   registerRouter(core)
   ...
}
```

就可以运行了。完整代码照旧放在 GitHub 的 geekbang/02 分支上了。



小结

今天，我们定义了一个属于自己框架的 Context，它有两个功能：在各个 Goroutine 间传递数据；控制各个 Goroutine，也就是是超时控制。

这个自定义 Context 结构封装了 net/http 标准库主逻辑流程产生的 Context，与主逻辑流程完美对接。它除了实现了标准库的 Context 接口，还封装了 request 和 response 的请求。你实现好了 Context 之后，就会发现它跟百宝箱一样，在处理具体的业务逻辑的时候，如果需要获取参数、设置返回值等，都可以通过 Context 获取。

封装后，我们通过三步走为请求设置超时，并且完美地考虑了各种边界场景。

你是不是觉得我们这一路要思考的点太多了，又是异常，又是边界场景。但是这里我要特别说明，其实真正要衡量框架的优劣，要看什么？就是看细节。

所有框架的基本原理和基本思路都差不多，但是在细节方面，各个框架思考的程度是不一样的，才导致使用感天差地别。所以如果你想完成一个真正生产能用得上的框架，这些边界场景、异常分支，都要充分考虑清楚。

思考题

在 context 库的官方文档中有这么一句话：

> Do not store Contexts inside a struct type;
> instead, pass a Context explicitly to each function that needs it.
> The Context should be the first parameter.

大意是说建议我们设计函数的时候，将 Context 作为函数的第一个参数。你能理解官方为什么如此建议，有哪些好处？可以结合你的工作经验，说说自己的看法。

欢迎在留言区分享你的思考，畅所欲言。如果你觉得今天的内容有所帮助，也欢迎你分享给你身边的朋友，邀他一起学习。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01｜net/http：使用标准库搭建Server并不是那么简单

下一篇 03｜路由：如何让请求更快寻找到目标函数？

精选留言 (5)

写留言

恶魔果实

2021-09-18

为什么会导致服务b和服务c的瞬时请求加大？这里不是很理解。
a请求失败，但是b，c请求是成功的呀。

展开

1

zhao
2021-09-17

代码里面context的封装跟gin的源代码真是像极了，对照来看，对框架的理解又加深了一些。

作者回复: 你好，是的context的封装和gin的代码很像，本质就是通过自己开发对开源的gin会有进一步理解。

1 1

qinsi
2021-09-15

“context作为函数的第一个参数”大概有两层意思。一是作为函数的参数传入。这个应该是针对在一个struct的多个方法中共享一个context的情况说的。因为每个方法都有可能需要创建子context，所以不应该共享而是应该显式传递。二是作为第一个参数。这个多半是一种约定。在支持可变长参数的语言中，固定参数只能出现在可变参数的前面。而作为与业务逻辑关系不大的context，出现在第一个的位置也方便也其他参数作区分。...

展开

作者回复: context作为第一个参数在实际工作中是非常有用的一个实践。不管我们是设计一个函数，或者设计一个结构体的方法，或者服务的时候，我们一旦养成了将第一个参数作为context的习惯，那么这个context在相互调用的时候，就会传递下去。这里会带来几个好处：

1 链路通用内容传递。context中是可以通过WithValue方法将某些字段封装在context里面，并且传递的。最常见的字段是traceId, spanId。而在日志中带上这些ID，再将日志收集起来，我们就能进行分析了。这也是我们现在比较流行的全链路分析的原理。

2 链路统一设置超时。我们在定义一个服务的时候，将第一个参数固定设置为context，则可以通过这个context进行超时设置，而这个超时设置，是由上游调用方进行设置，这样就形成了一个统一的超时设置机制。比如A设置了5s超时，自己使用了1s，传递到下游B服务的时候，设置B的context超时时长为4s。这样全链路超时传递下去，就能保持统一设置了。

是的，请求控制和请求实现混在一起的情况，后面引入middleware会改掉的。

1

zhao

2021-09-17

搞清楚这一点，我们回看第二步，做完具体业务逻辑就结束是不行的，还需要处理 panic。所以这个 Goroutine 应该要有两个 channel 对外传递事件。

这行话下面的代码块，go 一个匿名函数的时候少了右括号')'。

展开

编辑回复: 感谢反馈👍补充文件夹出处的时候把代码块拆开了，估计不小心拆掉了一个

.

2021-09-15

叶大可不可以在标注一下每段代码是在哪个文件夹里？从零开始的话不知道项目的目录结构是怎么样的

作者回复: 后面单开一章专门介绍目录怎么设计，现在我们先专注把实现写出来，拉分支放就行，每一章我都把源码放在github项目的对应分支上了，可以先跟着文章敲完代码，再跟着对应分支对一遍。